Reverse Engineering

Crusoe Exposed: Transmeta TM5xxx Architecture

Source: "Anonymous" on Real World Technologies websire

Part 1:
https://www.realworldtech.com/crusoe-intro/
Part 2:
https://www.realworldtech.com/crusoe-exposed/

Part 1

Background

This is the first in a series of articles that describes the reverse engineering of the internal ISA used by the Transmeta Crusoe microprocessor.

Ever since Transmeta introduced its Crusoe microprocessors in 1999, there has been a great deal of interest in the internal architecture of these novel devices, the Code Morphing Software (CMS) responsible for their IA-32/x86 compatibility, and their true performance potential if not hampered by dynamic binary translation overhead.

Unfortunately, Transmeta has repeatedly stated that the underlying Crusoe instruction set will not be publicly documented, and have even implied that reverse engineering the instruction set or CMS itself would be impossible.

It is particularly disturbing that key open source figures, including Linus Torvalds himself, have become involved with such a belief in 'security through obscurity' (see 4(https://www.realworldtech.com/crusoe-intro/4) in references for lkml quotes).

Unfortunately for Transmeta, as we will see below, this was a wildly incorrect and very presumptuous assumption. Given the appropriate technical background, skills and motivation, this report demonstrates that it is fully possible for someone outside of Transmeta's secret inner circle to reverse engineer not only the hardware instruction set but the operation of CMS itself.

In the interest of finally revealing what others have merely speculated on, the author has successfully reverse engineered a substantial part of the native Crusoe architecture and instruction set, right down to the binary instruction encodings and functional unit specifics for the TM5xxx series of processors.

It is important to note that absolutely no proprietary trade secrets were known a priori for this research, nor does the author have any relationship with the company or its employees. All information presented here is strictly the result of clean room reverse engineering over the course of the past few months.

Report Structure

This report is the first installment of a series of papers on Crusoe internals. The purpose of this initial release is to serve as a wake-up call to those at Transmeta and elsewhere who thought Crusoe was impervious to reverse engineering. It serves as the proof that the knowledge Transmeta sought to keep secret is in fact out there. Considering that this analysis was done for strictly academic reasons in my spare time, it is very reasonable to assume that Transmeta's full time competitors could possess similar knowledge.

The author is publishing this research anonymously so as to gauge the reaction to its release. If sufficient interest is present, a second report will be released in early 2004, which will describe in detail the instruction set, binary encodings, functional unit specifics and more. The author's disassembler and analysis tools source code will also be provided. A third

and final report will document all behavior specific to translated x86 code, as experimented with in real time on stock hardware.

The eventual goal is to allow developers to switch into native TM5xxx mode at runtime, allowing the modification of CMS itself. Contrary to statements released by Linus et al (see 4(https://www.realworldtech.com/crusoe-intro/4) in references) presumably to discourage this very effort, CMS does in fact contain 'back doors' to allow such modification. While the factory programmed CMS version in flash ROM does verify a DSA signature on any CMS upgrade images, there appears to be at least one backdoor that not even Transmeta itself was apparently aware of. As stated earlier, this will be presented in the next report once the author has time to write it up.

Reverse Engineering Methodology

Now, for the proof. The reverse engineering effort started with a static analysis of a CMS 4.4.03 binary image (5(https://www.realworldtech.com/crusoe-intro/4) in references). The only other resources used include published articles and technical reports, as well as the author's own personal experience in designing VLIW processors at a major university research lab.

Surprisingly, no actual Crusoe hardware was necessary for most of the initial static analysis; only isolated details required analyzing a CMS memory image at runtime. To present some idea of the nature of Crusoe native code, the end of this report includes disassembled and commented listings of a few key CMS functions. Each instruction is given in both binary and symbolic form, along with comments to demonstrate understanding of the actual high level meaning the corresponding source code expressed.

CMS appears to have been compiled from C source using a hacked version of gcc, with many key parts hand-scheduled in VLIW assembler. Note that the instruction mnemonics and register names may not exactly match what Transmeta developers use internally, since the author did not have access to that information. Hence, most symbolic names have been based on those used in published papers or were derived from debug strings in the CMS image.

Anyone from Transmeta should be able to verify that this is indeed genuine TM5800 code with the given meaning; they are invited to confirm the legitimacy of the reverse engineered code (assuming their employer will let them!)

References and Conclusions

This concludes the introductory installment of the Crusoe Exposed series. Watch for the next parts of this report to be posted in early 2004, and happy new year to Transmeta – we hope your TM8xxx Efficeon series will be even more interesting to reverse engineer.

References

[1] Klaiber, A. "The Technology Behind Crusoe Processors," http://www.transmeta.com/pdfs/paper_aklaiber_19jan00.pdf

[2] Halfhill, T. "Transmeta Breaks x86 Low-Power Barrier," Microprocessor Report, 14(2):9–18, February 2000.

[3] TM5800 BIOS Programmer's Guide: http://www.transmeta.com/crusoe_docs/TM5800_BIOSGuide_6-14-02.pdf

[4] Linux Kernel Mailing List: "Re: Crusoe's persistent translation on linux?", by Linus Torvalds, June 19, 2003. http://www.ussg.iu.edu/hypermail/linux/kernel/0306.2/1091.html http://www.ussg.iu.edu/hypermail/linux/kernel/0306.2/1221.html

[5] Code Morphing Software 4.4.03 Persistent Translation Technology Update for HP Tablet PCs. ftp://ftp.compaq.com/pub/softpaq/sp23501-24000/SP23689.exe

Handler for reading MSR registers from x86 operating system code. Illustrates the code for functionality well documented in published specifications from Transmeta.

read_msr:
#⇒ target 0×000a6798:
0×000a6798:
ALU0 5039aff4 = 010 100000011 100110 101111 11110100
ALU1 103befe0 = 000 100000011 101111 101111 11100000

addi %r38,%r47,-12 # %r38 = %sp – 12
addi %r47,%r47,-32 # %sp = %sp – 32

0×000a67a0:
LSU 03b00600 = 000 000111011 000000 000110 00000000
ALU1 1038e604 = 000 100000011 100011 100110 00000100
ALU0 1bb8bf00 = 000 110111011 100010 111111 00000000
imm 0×80000000

nop.lsu
addi %r35,%r38,4 # %r35 = %r38 + 4
oril %r34,%zero,0x80000000 # %r34 = 0x80000000

0×000a67b0:
LSU 044319bf = 000 001000100 001100 011001 10111111
ALU1 100f008b = 000 100000000 111100 000000 10001011
nop0 14b00000 = 000 101001011 000000 000000 00000000
imm 0×14b00000

st[%r47],%r25 # Save %r25
add %r60,%r0,%r34 # %r60 = %r0 (%eax) + 0x80000000
(to bring MSR offset down to zero base)

0×000a67c0:
LSU 04431a8f = 000 001000100 001100 011010 10001111
ALU1 09397c04 = 000 010010011 100101 111100 00000100
ALU0 18367f00 = 000 110000011 011001 111111 00000000
imm 0×0018aae0

st[%r35],%r26 # Save %r26

slli %r37,%r60,4 # Shift MSR number for indexing into table

addil%r25,%zero,0x0018aae0

0x18aae0 -> cpuid data for 0x80000000-6

NOTE: shifts (slli, etc). appear to only be available on ALU1, at least according to the opcode map. This is a bizarre (but low power) design.

0×000a67d0:
LSU 04431b9b = 000 001000100 001100 011011 10011011
ALU1 4606fa03 = 010 001100000 011011 111010 00000011
ALU0 1f3f8000 = 000 111110011 111110 000000 00000000
imm 0×80000006

st[%r38],%r27 # [%sp-12] = %r27 (callee saved)

001100000 %r27,%r58,%r0#

cmpil.c %sink,%r0,0x80000006 # compare %eax == 0x80000006?

0×000a67e0:
LSU 03b00600 = 000 000111011 000000 000110 00000000
ALU1 103ed9c0 = 000 100000011 111011 011001 11000000
ALU0 18392500 = 000 110000011 100100 100101 00000000
imm 0×0018aae0

nop.lsu

addi %r59,%r25,-64 # 0x18aaa0 -> cpuid data returned for 0x0-0x3

addil%r36,%r37,0x0018aae0 # %r36 = 0x18aae0 + (%r60 << 4)

0×000a67f0:
LSU 03b00600 = 000 000111011 000000 000110 00000000
ALU1 170f808b = 000 101110000 111110 000000 10001011
ALU0 9386bfff = 100 100111000 011010 111111 11111111
BRU ae014d0c = 101 0111 0 0000000101001101000011 00

nop.lsu

cmp.c%sink,%r0,%r34 # compare %eax == 0x80000000

or%r26,%zero,%zero # Move %r26 = 0

br.gt0x000a6860 # branch if %eax > 0x80000006

(to handle 0x80860000 functions)

0×000a6800:
ALU0 773f8003 = 011 101110011 111110 000000 00000011
BRU a6014d17 = 101 0011 0 000000010100110100010111

cmp.c%sink,%r0,3 # Compare %eax == 3

br.ge0x000a68b8 # branch if (%eax >= 0x80000000)

(branch to load_cpuid_data_to_regs)

0×000a6808:
ALU0 538f00ff = 010 100111000 111100 000000 11111111
ALU1 09394004 = 000 010010011 100101 000000 00000100

or%r60,%r0,%zero # %r60 = %eax

slli %r37,%r0,4 # %r37 = %eax << 4 (index cpuid table)

0×000a6810:
LSU 03b00600 = 000 000111011 000000 000110 00000000
ALU1 100925ef = 000 100000000 100100 100101 11101111
ALU0 93867bff = 100 100111000 011001 111011 11111111

nop.lsu

add %r36,%r37,%r59 # %r36 = %r37 + (%r0<<4) + 0x18aaa0

#⇒ target 0×000a6820:
0×000a6820:
LSU 03b00600 = 000 000111011 000000 000110 00000000
ALU1 1038ef14 = 000 100000011 100011 101111 00010100
ALU0 1380bfff = 000 100111000 000010 111111 11111111
imm 0×14b00000

nop.lsu

addi %r35,%r47,20# %r35 = %r47 + 20

or%r2,%zero,%zero# %edx = 0

0×000a6830:
ALU0 53807fff = 010 100111000 000001 111111 11111111
ALU1 1380ffff = 000 100111000 000011 111111 11111111

move %r1,%zero,%zero # %ecx = 0

move %r3,%zero,%zero # %ebx = 0

0×000a6838:
ALU0 53803fff = 010 100111000 000000 111111 11111111
ALU1 1038af18 = 000 100000011 100010 101111 00011000

move %r0,%zero,%zero # Set %eax = 0

addi %r34,%r47,24 # %r34 = %sp + 24

#⇒ target 0×000a6840:
0×000a6840:
LSU 2506c48f = 001 001010000 011011 000100 10001111
ALU1 46803a6f = 010 001101000 000000 111010 01101111

ld%r27,[%r35] # load %r27 = [%r35]

br.prep %r0,%r58,%r27 # prepare branch (%r58 = %link)

0×000a6848:
LSU 2506848b = 001 001010000 011010 000100 10001011
nop1 14b00000 = 000 101001011 000000 000000 00000000

ld%r26,[%r34] # restore saved register

nop

0×000a6850:
LSU 050644bf = 000 001010000 011001 000100 1011111
ALU1 103bef20 = 000 100000011 101111 101111 0010000
nop0 94b00000 = 100 101001011 000000 000000 00000000
BRU 80800076 = 100 0000 0 1000000000000000111011 0

ld%r25,[%r47] # restore saved register

addi %r47,%r47,32# restore stack frame

nop

br.ret %r58 # return to caller (%r58 = %from)

Load %eax/%ebx/%ecx/%edx with values for a given cpuid

MSR read request. Illustrates operation of load unit.

load_cpuid_data_to_regs:
#⇒ target 0×000a68b8:
0×000a68b8:
nop0 54b00000 = 010 101001011 000000 000000 00000000
nop1 14b00000 = 000 101001011 000000 000000 00000000

0×000a68c0:
ALU0 5039e40c = 010 100000011 100111 100100 00001100
ALU1 103a2404 = 000 100000011 101000 100100 00000100

addi %r39,%r36,12# %r39 = %r36 + 12(0x18aaec -> “smet”)

addi %r40,%r36,4 # %r40 = %r36 + 4 (0x18aae4 -> “Tran”)

0×000a68c8:
LSU 25000493 = 001 001010000 000000 000100 10010011
ALU1 1039a408 =<000>100000011 100110 100100 00001000

ld%r0,[%r36] # %eax = max cpuid request id

addi %r38,%r36,8 # %r38 = %r38 + 8 (= 0x18aae8)

0×000a68d0:
LSU 2500c4a3 = 001 001010000 000011 000100 10100011
ALU1 170f9aff = 000 101110000 111110 011010 11111111

ld%r3,4,[%r40]# load %ebx (%r3) = “Tran”

cmp.c%sink,%r26,%zero # (as called from above, %r26 always is zero)

0×000a68d8:
LSU 2500849f = 001 001010000 000010 000100 10011111
ALU1 1038ef14 = 000 100000011 100011 101111 00010100

ld%r2,[%r39] # load %edx (%r2) = “smet”

addi %r35,%r47,20# r35 = r47 + 20

0×000a68e0:
LSU 0500449b = 000 001010000 000001 000100 10011011
ALU1 1038af18 = 000 100000011 100010 101111 00011000
nop0 94b00000 = 100 101001011 000000 000000 00000000
BRU a8014d08 = 101 01000 00000001 01001101 00001000

ld%r1,[%r38] # load %ecx (%r1) = “aCPU” (%r38 = 0x18aae8)

addi %r34,%r47,24# %r34 = %r47 + 24

nop

br.eq0x000a6840 # branch to common completion

0×000a68f0:
LSU 03b00600 = 000 000111011 000000 000110 00000000
nop1 74b00014 = 011 101001011 000000 000000 00010100
nop0 14b00000 = 000 101001011 000000 000000 00000000
imm 0×14b00000

nop.lsu | nop | nop | nop

0x000a6900:
nop0 54b00000 = 010 101001011 000000 000000 00000000
nop1 74b00315 = 011 101001011 000000 000011 00010101
0x000a6908:
nop0 54b00000 = 010 101001011 000000 000000 00000000
nop1 74b00116 = 011 101001011 000000 000001 00010110
0x000a6910:
LSU 03b00600 = 000 000111011 000000 000110 00000000
nop1 74b00217 = 011 101001011 000000 000010 00010111
nop0 94b00000 = 100 101001011 000000 000000 00000000
BRU 80014d08 = 100 00000 00000001 01001101 00001000

nop.lsu | nop | nop

br0x000a6840

Called to enable/disable processor serial number (PSN)

when bit 21 of MSR 0x119 is written. Top level write_msr

handler is not shown, but is similar to read_msr.

On entry:

%r37 -> cpu_feature_flags (at 0x0018aabc)

%r38 -> current value of msr_psn_disable

#=> target 0x000a6f20:
0x000a6f20:
LSU 050a8497 = 000 001010000 101010 000100 10010111
ALU1 170f80ff = 000 101110000 111110 000000 11111111
ALU0 1a3b0000 = 000 110100011 101100 000000 00000000
imm 0xffdfffff

ld%r42,%r4,%r37/3# %r42 = [%r37] (load cpuid feature flags)

cmp.c%sink,%r0,%zero# is %eax zero? (%eax = 0) meaning: *enable* PSN

andil%r44,%r0,0xffdfffff # %r44 = %eax & ~(1<<21): bit21 = PSN disable bit

0x000a6f30:
LSU 03b00600 = 000 000111011 000000 000110 00000000
ALU1 020342ff = 000 000100000 001101 000010 11111111
ALU0 1a3a6a00 = 000 110100011 101001 101010 00000000
imm 0xfffbffff

nop.lsu

cmp.and %r13,%r2,%zero # %edx == %zero (high 32 bits in %edx must be zero)

andil%r41,%r42,0xfffbffff # 0xfffbffff = ~(1 << 18) = PSN feature flag

0×000a6f40:
LSU 03b00600 = 000 000111011 000000 000110 00000000
ALU1 1388e5ff = 000 100111000 100011 100101 11111111
ALU0 82036cff = 100 000100000 001101 101100 11111111
BRU a8014dc9 = 101 0100 0 0000000101001101110010 01

nop.lsu

or%r35,%r37,%zero# %r35 = %r37

cmp.and %r13,%r44,%zero# cmp.and %r44 == %zero

(i.e., no other bits are set in %eax)

br.eq0x000a6e48 # (if %eax = 0 [means "enable PSN"], branch)

0×000a6f50:
LSU 04432997 = 000 001000100 001100 101001 10010111
ALU1 13b9ff01 = 000 100111011 100111 111111 00000001
ALU0 103a25e4 = 000 100000011 101000 100101 11100100
imm 0×14b00000

st[%r37],%r41 # cpu_feature_flags &= PSN_FF_BIT

ori %r39,%zero,1# %r39 = 1

addi %r40,%r37,-28 # %r40 = 0x0018aaa0 [0x3 'Genu' 'Mx86' 'ineT']

0×000a6f60:
LSU 0443009b = 000 001000100 001100 000000 10011011
ALU1 10396518 = 000 100000011 100101 100101 00011000
ALU0 1039a514 = 000 100000011 100110 100101 00010100
imm 0×14b00000

st[%r38],%r0 # st [%r38],%r0 (%eax => msr_psn_disable)

addi %r37,%r37,24# %r37 = 0x18aad4 &("my unique id")

addi %r38,%r37,20# %r38 = 0x18aad0 &(before "my unique id")

0×000a6f70:
LSU 044327a3 = 000 001000100 001100 100111 10100011
nop1 14b00000 = 000 101001011 000000 000000 00000000

nop0 14b00000 = 000 101001011 000000 000000 00000000
imm 0×14b00000

st[%r40],%r39 # max_cpuid_base_func_number = %r39 (i.e., 1)

0×000a6f80:
LSU 24433f97 = 001 001000100 001100 111111 10010111
ALU1 1038a320 = 000 100000011 100010 100011 00100000

st[%r37],%zero# [“my u”] = 0

addi %r34,%r35,32# %r34 = 0x18aadc

0×000a6f88:
LSU 24433f9b = 001 001000100 001100 111111 10011011
ALU1 1039231c = 000 100000011 100100 100011 00011100

st[%r38],%zero# [word before “my unique id”] = 0

addi %r36,%r35,28# %r36 = 0x18aad8

0×000a6f90:
LSU 24433f8b = 001 001000100 001100 111111 10001011
nop1 14b00000 = 000 101001011 000000 000000 00000000

st[%r34],%zero# [“e ID”] = 0

0×000a6f98:
LSU 24433f93 = 001 001000100 001100 111111 10010011
nop1 14b00000 = 000 101001011 000000 000000 00000000

st[%r36],%zero# [“niqu”] = 0

0×000a6fa0:
nop0 54b00000 = 010 101001011 000000 000000 00000000
nop1 14b00000 = 000 101001011 000000 000000 00000000
0×000a6fa8:
nop0 74b00000 = 011 101001011 000000 000000 00000000
BRU 80014dc9 = 100 0000 0 00000001010011011100100 1

br0x000a6e48

Part 2

Introduction

This report is the second installment in the Crusoe Exposed series. It describes in detail the instruction set, binary encodings, functional unit specifics and more for the Transmeta TM5xxx microprocessors. Source code for a full disassembler and analysis toolset is provided. The first part can be found here.

Based on the response to the first report, I have decided to release substantially more details of my reverse engineering effort here. Fortunately, high level technical people at Transmeta have indicated that they have no legal intent to prevent the release of this information, but obviously cannot confirm nor deny its accuracy.

This is not the final report in this series, however hopefully the information provided here will be enough for other interested parties with more time on their hands to carry on my research into the dynamic behavior of CMS and the TM5xxx architecture while running x86 code.

As with all reports in this series, absolutely no proprietary trade secrets were used in this research. All the information presented here is strictly the result of clean room reverse engineering

This report is organized into three main sections. The first focuses on describing the TM5xxx architecture and microarchitecture, and the second is primarily observations, notes and critical analysis of Transmeta's work in the field of binary translation and dynamic optimization. The last section is the appendix, which includes source code for disassembling and analyzing CMS memory images, plus opcode maps, statistics and more.

Detailed Analysis of the TM5800 Microarchitecture and Instruction Set:

CMS Boot Procedure and Memory Map

At power up, the processor begins executing boot code from an internal ROM. After doing basic POST operations, the OEM configuration data (part of the flash ROM) is read via the processor's 4-pin flash ROM serial interface. This data tells the processor how to configure the system RAM and prepare for copying the main CMS image.

After the on-chip DRAM controller and caches are configured, the main CMS image in flash ROM is streamed into the core, decompressed and copied out to RAM. It will reside at physical address (total system RAM – 16MB). At this point, control is transferred to the main CMS image for further boot.

There are actually two copies of the CMS image in flash ROM. This is a failsafe mechanism: should the first copy fail to boot the processor (for instance, because it was corrupted by a power failure during an update), the second copy, called a "recovery ROM", is used on the next reboot. It is assumed that if CMS fails to set some internal watchdog register within a few million cycles after bootup, the primary ROM is assumed to be corrupted so the recovery ROM is used.

The decompressed CMS code image is actually mapped to offset 0×8e000 in CMS DRAM space, and takes up around 1.7 MB:

```
0x00000 to 0x8e000: occupied by CMS loader, etc. (not in CMS image)
0x8e000 to 0x1823f0: 1.0 MB CMS image
0x1823f0 to 0x1b2261: 196 KB CMS data
Above 0x1b2261 (~1.7 MB - 16 MB): uninitialized data, stack, heap and translation cache
```

CMS Image Format and Modification Techniques

The main CMS flash ROM image format, as found in cms4.4.03raptor.cru, consists of the following parts:

Header (192 bytes) with version information and other data
ROM image itself, compressed with the standard gzip deflate algorithm
DSA signature applied by Transmeta

The trailing signature is used to ensure only authenticated CMS updates are accepted by the update mechanism (accessible from x86 mode via various MSRs) before being written to the flash ROM. The DSA public key is contained in CMS around address 0×182654, while the private key is known only to Transmeta. This makes it impossible to load unsigned updates, at least if the normal update mechanism is used.

However, there is a back door: if we can alter the running CMS image in RAM, we can simply overwrite the public key with our own, or just patch the signature checking code to skip the verification step. Since the CMS private RAM is not accessible via x86 generated physical addresses, there are two methods of accessing it:

Using dedicated hardware (i.e., DRAM analyzer) or specialized system board with front side bus redirection circuitry. This is obviously not practical without physical access to the machine and appropriate hardware equipment.

Certain common peripheral chipsets (e.g., ethernet DMA controllers or IDE controllers) can be tricked into DMAing directly to/from CMS physical memory due to implementation bugs in the chipsets and the TM58xx processor. Again, this is not practical or possible on all Crusoe-based boards. It can also be very dangerous to wildly redirect DMAs to arbitrary bus addresses, unless you're an experienced hardware hacker.

Fortunately for Transmeta and its end users, this backdoor is difficult to exploit without the consent of the user, since it does require both x86 kernel level access and in some cases physical access to the machine. However, if you are experienced enough to be reading this, such limitations are unlikely to be a problem.

Processor Core Pipeline

The TM5xxx pipeline is mostly revealed in 4(https://www.realworldtech.com/crusoe-exposed/13), but here are further notes:

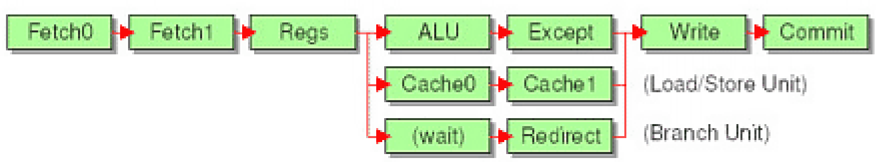

Figure 1 – Crusoe Pipeline Diagram

The top row of the diagram indicates the pipeline for an ALU instruction, with the other rows representing the two other types of logical units. The pipeline is a fairly typical RISC design:

Fetch0: The first 64 bits of a 64-bit or 128-bit bundle are fetched
Fetch1: The second 64 bits are fetched (for 128-bit bundles only)
Regs: Read source registers and decode/disperse instructions>
ALU: Execute single cycle operations in ALU0 and ALU1
Except: Complete two-cycle ALU0/ALU1 ops and detect exceptions
Cache0: Initiate L1 data cache access based on register address
Cache1: Complete L1 data cache access, TLB access and alias checks
Write: Write results back to GPRs or store buffer
Commit: Optionally latch the lower 48 GPRs into the shadow registers

Register Set

The processor has 64 GPRs, with the following specialized semantics:

- %r63 (%zero) always reads 0 when used as a source operand
- %r62 (%sink) is a discarded destination (e.g., for compares); it is never read
- %r59 (%from) saved return address
- %r58 (%link) return address
- %r47 (%sp) is the current stack pointer
- %r0 (%eax) for current x86 machine state
- %r1 (%ecx) for current x86 machine state
- %r2 (%edx) for current x86 machine state
- %r3 (%ebx) for current x86 machine state

The lower 48 of these GPRs are backed by shadowed GPRs: whenever a bundle has its commit bit set, the Commit stage latches the current values of the GPRs into the 'known good' shadow GPRs.

The processor also includes 32 80-bit floating point registers and 16 FP shadow registers; these are not discussed in this report.

There are also a wide variety of special purpose registers (SPRs), including the condition codes, profiling registers, power control settings and so on.

Instruction Encoding

Instructions are encoded in little endian byte and word order as shown in the following diagram:

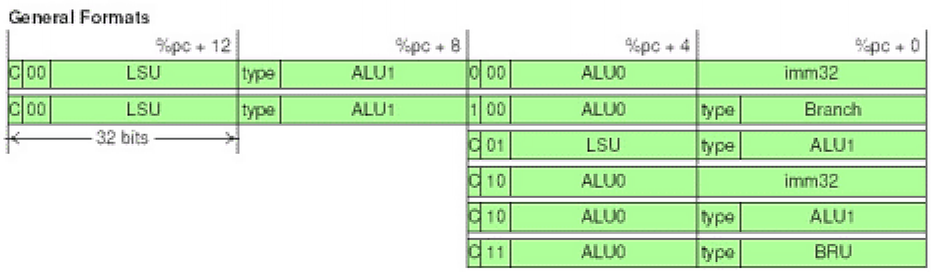

Figure 2 – Instruction Encoding for Crusoe

Instructions in both the ALU0 and ALU1 slots may have an 8-bit signed immediate instead of register rb; the ALU1 slot may optionally use a 32-bit immediate, but only in appropriate bundle types. All instructions (except branches) have a 9-bit opcode field. All opcodes share a common mapping into this 9-bit space, even though not all instructions can execute on all functional units.

The hardware is designed to interlock all operations through scoreboarding, however as described in the observations section, design flaws sometimes prevent the microprocessor from taking full advantage of these features.

The ALU0|imm32 and ALU0|ALU1 bundle types share the same format code (10) but the ALU1 slot is interpreted as an imm32 depending on the opcode:

11xxxx011: 32-bit immediate in place of ALU1

All others: execute ALU1 as instruction

If the 11xxxx011 pattern appears in an ALU1 slot, an 8-bit immediate is used instead. It is not clear why this encoding is sometimes used instead of the normal 8-bit immediate form.

ALUs

Two ALUs are provided in Crusoe. It appears that ALU1 executes a superset of the operations available on ALU0. In particular, shifts are available only on ALU1 (a rather bizarre decision given the amount of bit manipulation CMS does). It is unknown if ALU0 also implements simpler bit extraction operations. Figure 3 below shows the formatting for instructions to the two ALUs.

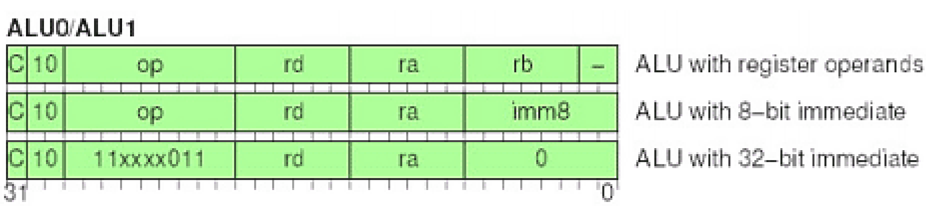

Figure 3 – Crusoe ALU Instruction Format

Both ALUs may have an 8-bit signed immediate instead of register rb; this is determined by the opcode (see the opcode map). The ALU0 slot may optionally use a 32-bit immediate, but only in appropriate bundle types, and only when the opcode matches the pattern 11xxxx011.

Condition codes exactly mirror the x86 semantics, right down to their encodings (see the branch unit section). Any instruction in either ALU0 or ALU1 can optionally latch the resulting condition codes if specified by the opcode (see opcode map). However, obviously only one of the two ALU0|ALU1 slots per bundle can write the condition codes in a single cycle. Condition codes cannot be generated on the same cycle as a dependent branch (unlike as in IA-64), but must be ready the cycle before.

There appears to be a facility for combining the existing condition codes with a comparison result, a la the PowerPC crand/cror instructions. This may be used via the cmpand (opcode 000100000) instruction et al.

The ALU1 slot is also used for all floating point and MMX operations, as indicated by ALU1's type select bits being something other than '00'. Since FP and MMX are obviously not used by the core CMS code, little is known about the FP/MMX unit at this point.

Load/Store Unit (LSU)

The single load/store unit performs all loads and stores, alias operations and various other memory related tasks. The instruction format is shown below in Figure 4.

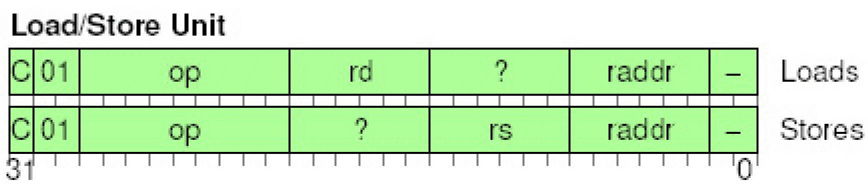

Figure 4 – Crusoe Load/Store Unit Instruction Format

All LSU operations take a fully calculated address in register ra; as with most VLIW architectures, no ra+offset or ra+rb addressing modes are provided.

Two kinds of loads and stores are possible: operations on physical CMS space addresses (as used in CMS itself), and operations as user code sees memory; i.e., addresses are translated by the TLB and can never access the protected CMS space.

The processor has two special 8KB SRAMs: the local program memory (LPM) and local data memory (LDM). The LPM holds often executed assist code for x86 page table lookups, alignment fixups, low level exception handling, interrupt handling, etc. This avoids having to bring such critical code into the L1 instruction cache on demand. The LDM contains data used by the LPM functions; i.e., copies of key x86 MSRs, native code stack, etc.

Branch Unit

Branches (both conditional and unconditional) within CMS use a 23 bit absolute target address aligned to a 64-bit boundary (i.e., abstarget is shifted left 3 bits). Interestingly, with 23 bits, 1 << (23+3) only allows up to 64 MB to be dedicated to CMS; this is probably one reason why larger translation caches have not been used.

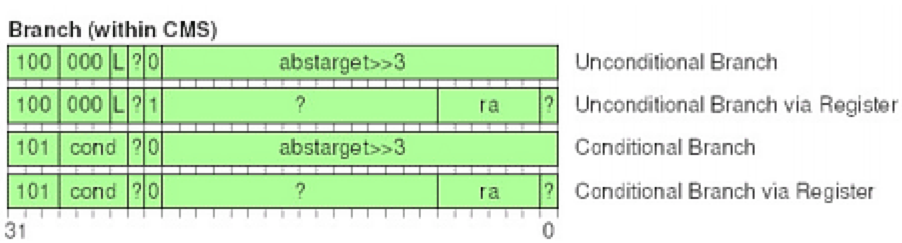

Figure 5 – Crusoe Branch Instruction Format

It appears that the CMS address space is the only region from which code can be executed; the processor is physically incapable of executing code directly from user space. This makes sense considering that all x86 code must be translated (and thus copied to CMS space) before native execution.

Conditional branches use the exact same condition code set (cc bits) as the x86 encoding in jump instructions (see the Intel manuals). Unconditional branches can optionally write the return address to the %link register (%r58) if the L bit (bit 0 of the cc field) is set.

Indirect branches occur through a general purpose register. It appears that special instructions are provided to prepare for an indirect branch when the target address is known in advance; this avoids the three-cycle branch penalty. In addition, special instructions may provide a branch with link functionality.

Observations, Notes and Critical Analysis of Transmeta's Work

Performance Issues

The scarcity of ALUs (only 2 are provided) appears to be a major impediment to the performance of both CMS itself and the generated code. Since all address generation is performed in these ALUs prior to forwarding the final virtual address to the LSU, the TM5xxx is at a clear disadvantage to the P6 and K7 microarchitectures (which have dedicated AGUs for each LSU). This appears to have been remedied in the just released TM8000 design, however.

Transmeta clearly also made major performance sacrifices in exchange for low power. For instance, as described in the ALU section, apparently only ALU1 can do shift operations. This can seriously impact some x86 code, not to mention the CMS translation code itself.

The instruction encoding also incurs a large number of wasted no-op slots: out of the roughly 231313 instructions in the CMS version examined, 70363 (over 30%) were nops. This could easily be remedied by using a stop bit based format as in TI DSPs or IA-64; I suspect the TM8000 has switched to such a format.

Programming Model

Most of the non time critical parts of CMS itself appear to be written in C, and compiled with a version of gcc and binutils hacked to generate TM5xxx code. This is evident from the relatively poor scheduling of certain code sequences, compared to what a good trace scheduler or programmer could do.

The released CMS image also appears to contain external debugger support, possibly accessible through the JTAG port. It is known that major hardware vendors have been given the appropriate debugging kits, so it may be possible to activate the CMS debug code externally.

Hardware Oddities

Even though the processor is fully interlocked via register operand score-boarding, there are many instances where delay slots were present in the disassembled code even though they are not strictly necessary nor present in other places. This is most likely the result of hardware bugs in the pipeline logic, which Transmeta has in fact admitted are masked by CMS in shipping hardware.

There are also a number of cases where instructions which could normally be paired together are not scheduled this way. It has been documented that CMS uses a different instruction mix at different clock speeds such that none of the critical circuit timing paths used by a given bundle will exceed the clock period.

The processor is typically clocked at the maximum frequency while running code within CMS itself, since this is infrequent yet obviously must be done as fast as possible. However, this does appear to limit the valid instruction combinations found in the CMS image to a subset of those possible when running at slower clock speeds.

Porting OS Kernels and Applications to the Underlying Hardware

Many insiders, including Linus Torvalds himself, have stated that it would be impossible to port an operating system kernel like Linux to the bare TM5xxx hardware, let alone user programs. The information learned from this reverse engineering effort does indeed seem to support this claim.

First of all, as discussed in the Branch Unit section, there is absolutely no hardware support for executing code directly from user memory. There is also no instruction TLB, and the L1 i-cache is physically mapped and tagged. Since all modern OS kernels (including Linux) assume that their own code pages are translated by the virtual memory mechanism, it would be impossible to port Linux to the bare Crusoe hardware without this capability.

There is also the issue of privileged instructions. As discussed in the Load Store Unit section, the TM58xx appears to support three main kinds of loads and stores: x86 in user mode, x86 in kernel mode and CMS physical space. The x86 protection scheme would fall apart if user applications are allowed to execute kernel context loads/stores and other privileged operations from user space, yet the underlying hardware provides no checking of this.

It appears that when CMS translates an x86 instruction trace, it will do so according to the context in which execution of the block was attempted (i.e., user mode or kernel mode). This may result in two separate versions of a given code block if executed from both the kernel and user space. Consequently, the translator will never generate privileged instructions in user code.

There is also the frequency scaling problem described in the Hardware Oddities section. User code targeting the native hardware ISA would have very restrictive scheduling of certain instructions without runtime awareness of the processor's unusual critical path restrictions.

Finally, it would be impossible to aggressively schedule loads and stores in native code without any corresponding sequentially ordered representation, i.e., x86 code, to fall back on in case of an exception. Without this, the exception handler would be unable to resolve the faulting instruction's dependency chain, since the hardware assumes CMS will interpret the block sequentially until the exception is found and resolved.

Origins

It should be noted that many of the ideas and internal implementations used in the TM32xx and TM5xxx chips were not even invented by Transmeta, despite what their patents appear to claim.

While I will not give a full analysis here, it appears that much of Transmeta's work was actually invented by IBM Research in the early 1990s. IBM's Daisy (Dynamically Architected Instruction Set from Yorktown) project 6(https://www.realworldtech.com/crusoe-exposed/13) is essentially CMS for the PowerPC architecture, and uses a strikingly similar design and implementation, including:

- Designing the morph host microarchitecture with the same semantics as the target instruction set (in IBM's case, PowerPC rather than x86)
- Translated page cache, using a T-bit buffer to track which user pages are dirty and need re-translation
- Explicit memory alias handling, using protected loads and checked stores
- Extensive profiling logic to aid in further optimization
- Handling of speculatively reordered loads and stores to I/O space

Even more similarities between CMS and IBM's work can be found in 8(https://www.realworldtech.com/crusoe-exposed/13) and 9(https://www.realworldtech.com/crusoe-exposed/13), which details BOA, a high clock speed VLIW successor to Daisy specifically intended for runtime binary translation of PowerPC systems.

Of course, Transmeta was still unquestionably the first to fully solve the thorny problem of transparent x86 binary translation in a commercially successful manner. With that in mind, many of the basic ideas expressed in Transmeta's patents appear to have substantial prior art in Daisy, and its successor, BOA 8(https://www.realworldtech.com/crusoe-exposed/13), 9(https://www.realworldtech.com/crusoe-exposed/13).

However, since BOA is a newer effort, it is not clear where the initial ideas came from. The current situation is that both IBM and Transmeta appear to have conflicting patents on the same technology, with the only difference being the x86 versus PowerPC specific aspects.

The similarities between Transmeta's and IBM's work is easy to prove, since the Daisy firmware has been released as open source 6(https://www.realworldtech.com/crusoe-exposed/13). IBM even acknowledges this apparent similarity in 7(https://www.realworldtech.com/crusoe-exposed/13), but notes that Transmeta has failed to provide any internal details to verify this claim 7(https://www.realworldtech.com/crusoe-exposed/13), 8(https://www.realworldtech.com/crusoe-exposed/13), 9(https://www.realworldtech.com/crusoe-exposed/13). Perhaps this report will shed some light on this mystery.

References

[1] Klaiber, A. "The Technology Behind Crusoe Processors," http://www.transmeta.com/pdfs/paper_aklaiber_19jan00.pdf

[2] Halfhill, T. "Transmeta Breaks x86 Low-Power Barrier," Microprocessor Report, 14(2):9–18, February 2000.

[3] TM5800 BIOS Programmer's Guide: http://www.transmeta.com/crusoe_docs/TM5800_BIOSGuide_6-14-02.pdf

[4] Linux Kernel Mailing List: "Re: Crusoe's persistent translation on linux?", by Linus Torvalds, June 19, 2003. http://www.ussg.iu.edu/hypermail/linux/kernel/0306.2/1091.html http://www.ussg.iu.edu/hypermail/linux/kernel/0306.2/1221.html

[5] Code Morphing Software 4.4.03 Persistent Translation Technology Update for HP Tablet PCs. ftp://ftp.compaq.com/pub/softpaq/sp23501-24000/SP23689.exe

[6] Daisy: Dynamically Architected Instruction Set from Yorktown, IBM Research. http://www.research.ibm.com/daisy/

[7] "How similar is DAISY to Transmeta?" http://www-124.ibm.com/developerworks/opensource/daisy/faq.html#transmeta

[8] Sathaye, S. et al. "BOA: Targeting Multi-Gigahertz with Binary Translation," IBM Research. http://www.research.ibm.com/vliw/Pdf/bt99.pdf

[9] Gschwind, M., Altman, E. "Inherently Lower Complexity Architectures using Dynamic Optimization," IBM Research. http://www.research.ibm.com/people/m/mikeg/papers/2002_wced.pdf

[10] Transmeta Patents

http://patft.uspto.gov, search for Transmeta under assignee name
http://appft.uspto.gov also has recent patent applications

Appendix A

Obtaining CMS Memory Images

The majority of the information in this report was derived from a static analysis of a CMS 4.4.03 binary image derived from 5(https://www.realworldtech.com/crusoe-exposed/13). If you download and extract all files from the Windows installer archive in 5(https://www.realworldtech.com/crusoe-exposed/13), there will be a cms4.4.03raptor.cru file. Use the following Linux commands to prepare this file according to the CMS Image Format section:

```
dd if=cms4.4.03raptor.cru of=cms4.4.03raptor.gz bs=192 skip=1
zcat cms4.4.03raptor.gz > cms4.4.03raptor.tmp
dd if=cms4.4.03raptor.tmp of=cms4.4.03raptor bs=1024 seek=568
```

The last command inserts 0×8e000 zero bytes before the final image to match its native logical base address, as described in the Memory Map section.

This gives the CMS binary image file cms4.4.03raptor that is used for all further static analysis.

Alternately, CMS can be dumped from a running Crusoe system as described in the CMS Image Format section; however, that method is much more difficult unless runtime analysis is required.

TM58xx Disassembler and Analysis Software

A full disassembler and analysis toolset for TM58xx CMS images is provided in source form as CrusoeReport-Part2-Disasm.cpp. This program was designed and tested under Linux only. It will take an appropriately decompressed and padded CMS binary image (see previous section) and disassemble it, complete with basic block analysis and statistical report generation. Both of these features are useful for further reverse engineering.

Sample Code Sequences

To present some idea of the nature of Crusoe native code, the included file CrusoeReport-Part2-Samples.S contains disassembled and commented listings of a few key CMS functions. Each instruction is given in both binary and symbolic form, along with comments to demonstrate understanding of the actual high level meaning the corresponding source code expressed.

Appendix B

Statistics for CMS image cms4.4.03raptor

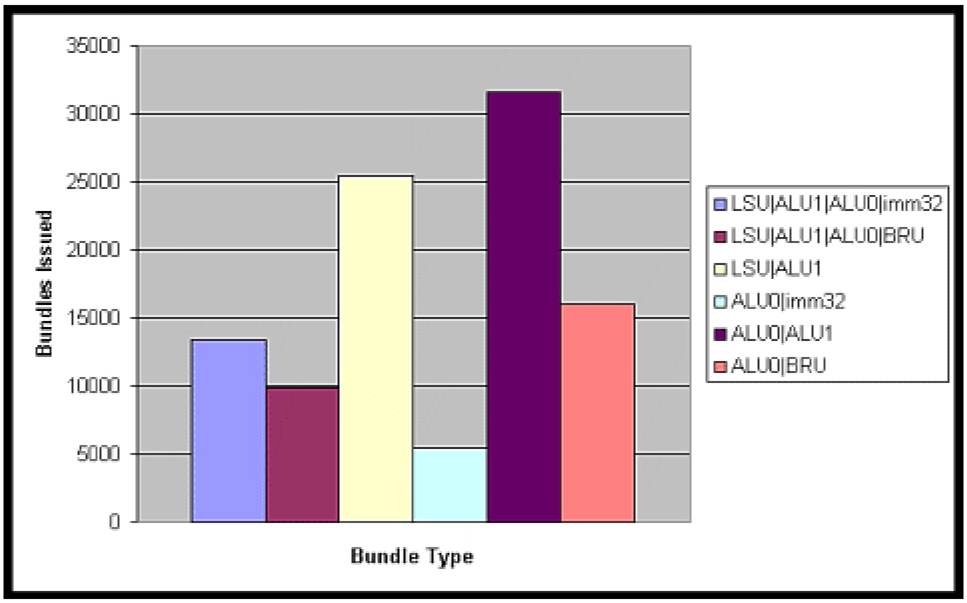

Figure 6 – Bundle Issue Chart

Total Instructions	231313	
Total Nops	70363	0.304189561
Bundles	Count	Percentage
LSU|ALU1|ALU0|imm32	13386	0.13154869
LSU|ALU1|ALU0|BRU	9911	0.097398705
LSU|ALU1	25401	0.249624104
ALU0|imm32	5409	0.053156048
ALU0|ALU1	31625	0.31078943
ALU0|BRU	16025	0.157483023
Total	101757	

Chart 1 – Bundle Frequency and nop Count

IPC figures are not provided since this obviously depends on the dynamic behavior of the code. With that in mind, most bundles appear to contain only two instructions; the quad-issue format is possible in only 22% of the code, primarily due to the limitations previously identified in the hardware.

Opcode Map and Instruction Descriptions

See the insn_name_table structure in CrusoeReport-Part2-Disasm.cpp for a map of the most common instruction opcodes and mnemonics. These instructions make up the vast majority of the CMS code.

Detailed Per-Unit Opcode Usage Statistics

The frequency of 9-bit opcodes used with different functional units is useful for determining the meaning and operation of the corresponding instructions.

The included CrusoeReport-Part2-histograms.txt file contains this data for each functional unit and bundle slot. The high 6 bits of the 9 bit opcode are in the rows, while the low 3 bits are in the columns.

CrusoeReport-Part2-Disasm.cpp

`/*

- Transmeta TM5xxx disassembler
-
- This is a full disassembler and analysis toolset for TM58xx CMS images.
- This program was designed and tested under Linux only. It will take an
- appropriately decompressed and padded CMS binary image (see "Obtaining
- CMS Memory Images" in the accompanying report) and disassemble it,
- complete with basic block analysis and statistical report generation.
-
- This program is licensed under the GNU General Public License.
  */
  #include <stdio.h>
  #include <stdlib.h>
  #include <errno.h>
  #include <string.h>
  #include <assert.h>

#define byte char
#define W32 unsigned long int
#define W32s signed long int
#define W64 unsigned long long int

#define bits(x, i, l) (((x) >> (i)) & ((1 << (l))-1))
#define bit(x, i) bits((x), (i), 1)

static inline W32s signext(W32s x, const int i) { return (x << (32-i)) >> (32-i); }
static inline W32s bitsext(W32s x, const int i, const int l) { return signext(bits(x, i, l), l); }

inline W32 hi32(W64 x) { return (W32)(x >> 32LL); }
inline W32 lo32(W64 x) { return (W32)(x & 0xffffffffLL); }

char *binstr(char* buf, W32 x, int n) {
for (int i = 0; i < n; i++) {
buf[(n-1)-i] = ((int)((x >> i) & 1)) + '0';

}
buf[n] = 0;
return buf;
}

char* binstr(W32 x, int n) {
static char buf[33];
return binstr(buf, x, n);
}

typedef char binstrbuf_t[64];

/*

- TM5xxx/TM32xx instruction format
  */
  typedef union Insn {
  struct { W32 unknown:2, rb:6, ra:6, rd:6, op:9, type:3; } alu;
  struct { W32 imm:8, ra:6, rd:6, op:9, type:3; } aluimm8;
  struct { W32 unknown:2, rb:6, ra:6, rd:6, op:9, type:3; } lsu;
  struct { W32 offset:23, ind:1, unknown1:1, cond:4, type:3; } br;
  struct { W32 unknown3:1, ra:6, unknown2:16, ind:1, unknown1:1, cond:4, type:3; } brreg;
  W32 imm;
  Insn(W32 data) { imm = data; }
  operator W32() const { return imm; }
  } Insn;

int lsu_insns_in_a_row = 0;
int bru_insns_in_a_row = 0;
int last_bru_insn_cond = 0;
int saved_bru_insn_cond = 0;

bool alu0_used, alu1_used, lsu_used, bru_used;

static const char* regnames[64] = {
"%r0", "%r1", "%r2", "%r3", "%r4", "%r5", "%r6", "%r7",
"%r8", "%r9", "%r10", "%r11", "%r12", "%r13", "%r14", "%r15",
"%r16", "%r17", "%r18", "%r19", "%r20", "%r21", "%r22", "%r23",
"%r24", "%r25", "%r26", "%r27", "%r28", "%r29", "%r30", "%r31",
"%r32", "%r33", "%r34", "%r35", "%r36", "%r37", "%r38", "%r39",
"%r40", "%r41", "%r42", "%r43", "%r44", "%r45", "%r46", "%sp",
"%r48", "%r49", "%r50", "%r51", "%r52", "%r53", "%r54", "%r55",
"%r56", "%r57", "%link", "%from", "%r60", "%r61", "%sink", "%zero"
};

#define _ 0

/*

- Primary opcode map:
- - high 6 bits in rows, low 3 bits in columns
- - '0' = used but as yet unknown
- - '_' = unused in CMS image
- - 'xxxi' = uses 8-bit immediate

- - 'xxxil' = uses 32-bit immediate
- - 'xxx.c' = sets condition codes
    /
    static const char insnname_table[1<<9] = {
    // 000 001 010 011 100 101 110 111
    /000000/ 0, 0, , , , , , ,
    /000001/ 0, 0, 0, 0, , , 0, 0,
    /000010/ 0, , 0, 0, 0, , , ,
    /000011/ 0, , 0, 0, , , , ,
    /000100/ "cmpand", 0, , 0, , , , ,
    /000101/ 0, 0, , 0, 0, , , ,
    /000110/ 0, 0, , 0, , , , ,
    /000111/ 0, 0, 0, "lsunop", , , , ,
    /001000/ 0, , , 0, "st", , , ,
    /001001/ 0, , , 0, 0, , , ,
    /001010/ "ld", , , 0, 0, , , ,
    /001011/ 0, , , 0, 0, , , ,
    /001100/ 0, , , 0, 0, , , ,
    /001101/ "movlink", , , 0, 0, , , ,
    /001110/ 0, , , 0, 0, , , ,
    /001111/ 0, , , 0, 0, , , ,

/010000/ 0, , , 0, 0, , , ,
/010001/ 0, 0, 0, 0, 0, , , ,
/010010/ 0, 0, 0, "slli", 0, 0, 0, 0,
/010011/ 0, 0, 0, 0, 0, , , 0,
/010100/ 0, 0, , 0, 0, , , ,
/010101/ , 0, 0, 0, 0, , , 0,
/010110/ 0, , , , 0, , , ,
/010111/ 0, , , , 0, , , ,
/011000/ 0, , , , , 0, 0, 0,
/011001/ 0, , , 0, , , , ,
/011010/ 0, 0, 0, 0, 0, 0, 0, 0,
/011011/ , , , , 0, , , ,
/011100/ 0, , , , , , , ,
/011101/ 0, , , 0, , , , ,
/011110/ 0, , , , , 0, 0, 0,
/011111/ , , , , , , , _,

/100000/ "add?", 0, 0, "addi", 0, 0, 0, 0,
/100001/ 0, , , 0, 0, , , ,
/100010/ 0, , , 0, 0, , , ,
/100011/ "oprr", 0, 0, 0, 0, , , ,
/100100/ "and", 0, 0, "andi", 0, 0, , 0,
/100101/ 0, , , , 0, , , ,
/100110/ 0, , , 0, 0, 0, 0, 0,
/100111/ "move", 0, 0, "ori", 0, , , 0,
/101000/ 0, , , 0, 0, , , ,
/101001/ 0, , , "nop", 0, , , ,
/101010/ 0, , , 0, 0, , , ,
/101011/ 0, , , , 0, , , ,
/101100/ 0, 0, 0, "?i.c", 0, , , ,

```
/101101/ 0, , , 0, 0, , , ,
/101110/ "sub.c", 0, 0, "cmpi.c", 0, , , ,
/101111/ 0, , , 0, , , , ,

/110000/ , , 0, "addil", , , 0, ,
/110001/ 0, , , , , , , ,
/110010/ 0, , , "?il", , , , ,
/110011/ 0, 0, 0, "?il.c", 0, , , ,
/110100/ 0, 0, 0, "andil", 0, , , 0,
/110101/ "or?", "?i", 0, "?il", 0, , , ,
/110110/ 0, , , "?il.c", 0, , 0, 0,
/110111/ 0, 0, 0, "moveil", 0, , , ,
/111000/ 0, , , , 0, , , ,
/111001/ 0, , , "?il", 0, , , ,
/111010/ 0, , , , 0, , , ,
/111011/ 0, , , , 0, , , ,
/111100/ 0, , 0, "?il.c", , , , ,
/111101/ 0, , , 0, 0, , , ,
/111110/ 0, , 0, "cmpil.c", , , , ,
/111111/ 0, , , 0, , , _, 0,
};

#undef _

#define OP_ALU_NOP 0×14b
#define OP_LSU_NOP 0×3b

int opcode_histogram_alu0[1<<9];
int opcode_histogram_alu1[8][1<<9];
int opcode_histogram_lsu[1<<9];

int total_insns = 0;
int total_nops = 0;

#define OP_USES_IMM32_MASK 0×187 // 11xxxx111
#define OP_USES_IMM32_VALUE 0×183 // 11xxxx011
inline bool op_uses_imm32(W32 op, int aluid) { return ((op & OP_USES_IMM32_MASK) == OP_USES_IMM32_VALUE) &&
(aluid == 0); }

void print_alu_insn(W32 insnbits, int aluid, W32 imm32 = 0) {
binstrbuf_t typestr, opcodestr, opcodestr2, rdstr, rastr, rbimmstr;
Insn insn = insnbits;
bool is_nop = (insn.alu.op == OP_ALU_NOP);
total_insns++;
if (is_nop) total_nops++;
bool has_imm32 = op_uses_imm32(insn.alu.op, aluid);

if (aluid == 0)
opcode_histogram_alu0[insn.alu.op]++;
else if (aluid == 1)
opcode_histogram_alu1[insn.alu.type][insn.alu.op]++;

printf(" %s%d %08x = %s %s %s %s %s # ", (is_nop) ? "nop" : "ALU", aluid, insn.imm,
binstr(typestr, insn.alu.type, 3),
binstr(opcodestr, insn.alu.op, 9),
```

binstr(rdstr, insn.alu.rd, 6),
binstr(rastr, insn.alu.ra, 6),
binstr(rbimmstr, insn.aluimm8.imm, 8));

if (is_nop) {
printf("nop\n");
} else {
char opnamebuf[16];

```
if (insn_name_table[insn.alu.op])
  snprintf(opnamebuf, sizeof(opnamebuf), "%s", insn_name_table[insn.alu.op]);
else snprintf(opnamebuf, sizeof(opnamebuf), "%s", binstr(insn.alu.op, 9));

printf("%-13s", opnamebuf);

printf("%s,%s,", regnames[insn.alu.rd], regnames[insn.alu.ra]);

if (has_imm32) {
  printf("0x%08x\n", imm32);
} else {
  printf("%s/%d\n", regnames[insn.alu.rb], signext(insn.aluimm8.imm, 8));
}
```

}

alu0_used = !is_nop;
}

void print_alu0_insn(W32 insn, W32 imm32 = 0) {
print_alu_insn(insn, 0, imm32);
}

void print_alu1_insn(W32 insn) {
print_alu_insn(insn, 1);
}

void print_lsu_insn(Insn insn) {
binstrbuf_t typestr, opcodestr, rdstr, rastr, rbimmstr;
bool is_nop = (insn.lsu.op == OP_LSU_NOP);
total_insns++;
if (is_nop)
total_nops++;
opcode_histogram_lsu[insn.lsu.op]++;

printf(" %s %08x = %s %s %s %s %s # ", (is_nop) ? "nopl" : "LSU ", insn.imm,
binstr(typestr, insn.lsu.type, 3),
binstr(opcodestr, insn.lsu.op, 9),
binstr(rdstr, insn.lsu.rd, 6),
binstr(rastr, insn.lsu.ra, 6),
binstr(rbimmstr, insn.aluimm8.imm, 8));
if (is_nop) {
printf("nop.lsu\n");
} else {
char opnamebuf[16];

```
if (insn_name_table[insn.lsu.op])
  snprintf(opnamebuf, sizeof(opnamebuf), "%s", insn_name_table[insn.lsu.op]);
else snprintf(opnamebuf, sizeof(opnamebuf), "%s", binstr(insn.lsu.op, 9));

printf("%-13s", opnamebuf);

printf("%s,%s,[%s]\n", regnames[insn.lsu.rd], regnames[insn.alu.ra], regnames[insn.lsu.rb]);
```

}

lsu_used = !is_nop;
}

// see Intel manual, p861:
static const char* br_cond_name[16] = {
"o", "no", "lt", "ge",
"eq", "ne", "le", "gt",
"s", "ns", "pe", "po",
"lt.s", "ge.s", "le.s", "gt.s",
};

void print_bru_insn(Insn insn) {
binstrbuf_t typestr, condstr, offsetstr, unknownstr, regstr;
char insnstr[64];

printf(" BRU %08x = %s %s %s %s ", insn.imm, binstr(typestr, insn.br.type, 3), binstr(condstr, insn.br.cond, 4), (insn.br.unknown1) ? "1" : "0", (insn.br.ind) ? "1" : "0");
if (insn.br.ind) {
printf("%s %s %s # ", binstr(unknownstr, insn.brreg.unknown2, 16), binstr(regstr, insn.brreg.ra, 6), insn.brreg.unknown3 ? "1" : "0");
} else {
printf("%s # ", binstr(offsetstr, insn.br.offset, 23));
}

switch (insn.br.type) {
case 4: // 0b100: unconditional branch
snprintf(insnstr, sizeof(insnstr), "br%s%s", bit(insn.br.cond, 1) ? ".link" : "", (insn.br.unknown1) ? ".u1" : "");
break;
case 5: // 0b101: conditional branch
snprintf(insnstr, sizeof(insnstr), "br.%s%s", br_cond_name[insn.br.cond], (insn.br.unknown1) ? ".u1" : "");
break;
default:
snprintf(insnstr, sizeof(insnstr), "br.%d%s", insn.br.type, (insn.br.unknown1) ? ".u1" : "");
break;
}

if (insn.br.ind) {
printf("%-12s %s", insnstr, regnames[insn.brreg.ra]);
if (insn.brreg.unknown3) printf(",u3");
if (insn.brreg.unknown2) printf(",%s", binstr(offsetstr, insn.brreg.unknown2, 16));
printf("\n");
} else {
printf("%-12s 0x%08x\n", insnstr, (insn.br.offset << 3));
}

```
total_insns++;
last_bru_insn_cond = insn.br.cond;
if (insn.br.type == 5) bru_used = true;
}

#define CODE_SEG_START 0x8e000
#define DATA_SEG_START 0x1823f0

#define BUNDLE_TYPE_LSU_ALU1_ALU0_IMM32 0
#define BUNDLE_TYPE_LSU_ALU1_ALU0_BRU 1
#define BUNDLE_TYPE_LSU_ALU1 2
#define BUNDLE_TYPE_ALU0_IMM32 3
#define BUNDLE_TYPE_ALU0_ALU1 4
#define BUNDLE_TYPE_ALU0_BRU 5
int bundle_type_histogram[6];

#define COLBITS 3

void print_opcode_histogram(const char name, int stats) {
printf("%s\n", name);
printf(" ");
for (int i = 0; i < (1<<COLBITS); i++) {
binstrbuf_t lowbits;
printf("%-10s ", binstr(lowbits, i, COLBITS));
}
printf("\n");

for (int i = 0; i < (1<<9); i++) {
if (bits(i, 0, COLBITS) == 0) {
binstrbuf_t highbits;
printf(" %9s: ", binstr(highbits, bits(i, COLBITS, 9-COLBITS), 9-COLBITS));
}
printf("%-10d ", stats[i]);
if (bits(i, 0, COLBITS) == (1<<COLBITS)-1) {
printf("\n");
}
}
printf("\n");
}

int main(int argc, char argv[]) {
FILE fd = fopen(argv[1], "r");
fseek(fd, 0, SEEK_END);
int count = ftell(fd);
fseek(fd, 0, SEEK_SET);
W32 buf = (W32)malloc(count);
assert(fread(buf, count, 1, fd) == 1);
fclose(fd);

W32 startinsn = (argc > 2) ? (W32)(atoi(argv[2]) / 4) : 0;

count /= 4;
printf("Found %d instructions, buffer at 0x%08x\n", count, buf);
printf("Starting at offset 0x%08x (insn 0x%08x)\n", startinsn*4, startinsn);
```

byte* branch_target_map = new byte[count/2];
memset(branch_target_map, 0, count/2);

```
W32* p;
```

W32 ip;

//
// Find branch targets
//
p = &buf[startinsn];
ip = 0;

while ((p < &buf[count])) {
if (ip < CODE_SEG_START) {
p += 4;
ip += 16;
continue;
}
if (ip >= DATA_SEG_START) {
break;
}

```
W64 insn = *(W64*)p;
int type = bits(insn, 61, 2);
int commit = bit(insn, 63);

switch (type) {
        case 0: {
  p += 4;
  ip += 16;
  if (commit) {
    Insn brinsn = (Insn)lo32(insn);
    W32 branch_target = brinsn.br.offset << 3;
    if ((branch_target>>3) < (count/2)) branch_target_map[branch_target>>3] = 1;
  }
  break;
}
case 1: { // LSU|ALU1
  p += 2; ip += 8; break;
}
case 2: { // ALU0|ALU1
  p += 2; ip += 8; break;
}
case 3: { // ALU0|BRU
  Insn brinsn = (Insn)lo32(insn);
  W32 branch_target = brinsn.br.offset << 3;
  if ((branch_target>>3) < (count/2)) branch_target_map[branch_target>>3] = 1;
  p += 2; ip += 8;
  break;
}
}
```

```
}

//
// Print the instructions
//
p = &buf[startinsn];
ip = 0;
while (p < &buf[count]) {
if (ip < CODE_SEG_START) {
p += 4;
ip += 16;
continue;
}
```

```
if (ip >= DATA_SEG_START) {
  break;
}

W64 insn = *(W64*)p;
int type = bits(insn, 61, 2);
int commit = bit(insn, 63);

if (branch_target_map[(ip >> 3)]) {
  printf("#=> target 0x%08x:\n", ip);
}

printf("0x%08x:\n", ip);

alu0_used = false;
alu1_used = false;
lsu_used = false;
bru_used = false;

switch (type) {
        case 0: {
  W64 insn2 = *((W64*)&p[2]);
  p += 4;
  ip += 16;
  print_lsu_insn(hi32(insn2));
  print_alu1_insn(lo32(insn2));
  print_alu0_insn(hi32(insn), lo32(insn));
  if (commit) {
    print_bru_insn(lo32(insn));
    bundle_type_histogram[BUNDLE_TYPE_LSU_ALU1_ALU0_BRU]++;
  } else {
    printf("  imm  0x%08x\n", lo32(insn));
    bundle_type_histogram[BUNDLE_TYPE_LSU_ALU1_ALU0_IMM32]++;
  }
  if (bit(hi32(insn2), 31)) printf("  commit\n");
  break;
}
case 1: { // LSU|ALU1
  Insn insn0 = (Insn)hi32(insn); // (alu0 insn)
  Insn insn1 = (Insn)lo32(insn);
  print_lsu_insn(insn0);
  print_alu1_insn(insn1);
  bundle_type_histogram[BUNDLE_TYPE_LSU_ALU1]++;
  if (commit) printf("  commit\n");
  p += 2; ip += 8;
  break;
}
case 2: { // ALU0|ALU1 or ALU0+imm32
  Insn insn0 = (Insn)hi32(insn); // (alu0 insn)
  Insn insn1 = (Insn)lo32(insn); // (alu1 insn)
  bool has_imm32 = op_uses_imm32(insn0.alu.op, 0);
  if (has_imm32) {
    print_alu0_insn(insn0, insn1.imm);
```

```
      printf("  imm  0x%08x\n", insn1.imm);
      bundle_type_histogram[BUNDLE_TYPE_ALU0_IMM32]++;
    } else {
      print_alu0_insn(insn0);
      print_alu1_insn(insn1);
      bundle_type_histogram[BUNDLE_TYPE_ALU0_ALU1]++;
    }
    if (commit) printf("  commit\n");
    p += 2; ip += 8;
    break;
  }
  case 3: { // ALU0|BRU
    Insn insn0 = (Insn)hi32(insn); // (alu0 insn)
    Insn insn1 = (Insn)lo32(insn);
    print_alu0_insn(insn0);
    print_bru_insn(insn1);
    bundle_type_histogram[BUNDLE_TYPE_ALU0_BRU]++;
    p += 2; ip += 8;
    if (commit) printf("  commit\n");
    break;
  }
  }

  if (lsu_used) {
    lsu_insns_in_a_row++;
    if (lsu_insns_in_a_row >= 32) {
      printf("# %d LSU insns in sequence\n", lsu_insns_in_a_row);
    }
  } else {
    lsu_insns_in_a_row = 0;
  }

  if (bru_used) {
    if (bru_insns_in_a_row == 0)
      saved_bru_insn_cond = last_bru_insn_cond;

    if (last_bru_insn_cond == saved_bru_insn_cond) {
      bru_insns_in_a_row++;
      if (bru_insns_in_a_row >= 2) {
        printf("# %d BRU insns in sequence\n", bru_insns_in_a_row);
      }
    } else {
      bru_insns_in_a_row = 0;
    }
  } else {
    bru_insns_in_a_row = 0;
  }

  }
```

printf("\n");
printf("Total instructions: %15d\n", total_insns);
printf("Total nops: %15d\n", total_nops);

```
printf("\n");
printf("Bundle type histogram:\n");
printf("LSU|ALU1|ALU0|imm32: %15d\n", bundle_type_histogram[BUNDLE_TYPE_LSU_ALU1_ALU0_IMM32]);
printf("LSU|ALU1|ALU0|BRU: %15d\n", bundle_type_histogram[BUNDLE_TYPE_LSU_ALU1_ALU0_BRU]);
printf("LSU|ALU1: %15d\n", bundle_type_histogram[BUNDLE_TYPE_LSU_ALU1]);
printf("ALU0|imm32: %15d\n", bundle_type_histogram[BUNDLE_TYPE_ALU0_IMM32]);
printf("ALU0|ALU1: %15d\n", bundle_type_histogram[BUNDLE_TYPE_ALU0_ALU1]);
printf("ALU0|BRU: %15d\n", bundle_type_histogram[BUNDLE_TYPE_ALU0_BRU]);

print_opcode_histogram("Opcode histogram for ALU0", opcode_histogram_alu0);
print_opcode_histogram("Opcode histogram for ALU1, type 0", opcode_histogram_alu1[0]);
print_opcode_histogram("Opcode histogram for ALU1, type 1", opcode_histogram_alu1[1]);
print_opcode_histogram("Opcode histogram for ALU1, type 2", opcode_histogram_alu1[2]);
print_opcode_histogram("Opcode histogram for ALU1, type 3", opcode_histogram_alu1[3]);
print_opcode_histogram("Opcode histogram for ALU1, type 4", opcode_histogram_alu1[4]);
print_opcode_histogram("Opcode histogram for ALU1, type 5", opcode_histogram_alu1[5]);
print_opcode_histogram("Opcode histogram for ALU1, type 6", opcode_histogram_alu1[6]);
print_opcode_histogram("Opcode histogram for ALU1, type 7", opcode_histogram_alu1[7]);
print_opcode_histogram("Opcode histogram for LSU", opcode_histogram_lsu);
}
```

Crusoe-Part2-histograms.txt

```
Opcode histogram for ALU0
000 001 010 011 100 101 110 111
000000: 66 0 0 0 0 0 0 0
000001: 12 20 3 3 0 0 1 24
000010: 0 0 23 0 0 0 0 0
000011: 0 0 9 1 0 0 0 0
000100: 353 13 0 120 0 0 0 0
000101: 360 2 0 62 2 0 0 0
000110: 368 2 0 2 0 0 0 0
000111: 49 2 6 33 0 0 0 0
001000: 0 0 0 25 0 0 0 0
001001: 0 0 0 0 0 0 0 0
001010: 0 0 0 1 0 0 0 0
001011: 0 0 0 0 0 0 0 0
001100: 20 0 0 1 0 0 0 0
001101: 0 0 0 2 0 0 0 0
001110: 15 0 0 1 0 0 0 0
001111: 0 0 0 0 0 0 0 0
010000: 0 0 0 18 0 0 0 0
010001: 0 0 0 0 0 0 0 0
010010: 0 0 0 0 0 0 0 0
010011: 0 0 0 0 0 0 0 0
010100: 0 0 0 0 0 0 0 0
010101: 0 0 0 0 0 0 0 0
010110: 0 0 0 0 0 0 0 0
010111: 0 0 0 0 0 0 0 0
011000: 0 0 0 0 0 0 8 8
011001: 528 0 0 448 0 0 0 0
011010: 0 4 5 3 0 36 91 253
011011: 0 0 0 0 0 0 0 0
011100: 2 0 0 0 0 0 0 0
```

011101: 0 0 0 0 0 0 0 0
011110: 0 0 0 0 0 1 9 9
011111: 0 0 0 0 0 0 0 0
100000: 589 26 77 5882 26 1 0 7
100001: 3 0 0 1 3 0 0 0
100010: 55 0 0 21 0 0 0 0
100011: 59 3 0 25 3 0 0 0
100100: 219 48 10 1158 40 4 0 217
100101: 1 0 0 0 3 0 0 0
100110: 327 0 0 28 971 321 21 374
100111: 6920 300 36 2530 14 0 0 3
101000: 277 0 0 9 0 0 0 0
101001: 1016 0 0 34743 0 0 0 0
101010: 23 0 0 4 0 0 0 0
101011: 0 0 0 0 0 0 0 0
101100: 79 2 2 689 0 0 0 0
101101: 0 0 0 0 0 0 0 0
101110: 1698 177 44 818 0 0 0 0
101111: 39 0 0 3 0 0 0 0
110000: 0 0 79 6087 0 0 2 0
110001: 0 0 0 0 0 0 0 0
110010: 0 0 0 0 0 0 0 0
110011: 0 0 1 5 0 0 0 0
110100: 0 0 94 1402 0 0 0 404
110101: 0 0 0 0 0 0 0 0
110110: 0 0 0 4 0 0 10 187
110111: 0 0 18 3558 0 0 0 0
111000: 328 0 0 0 0 0 0 0
111001: 0 0 0 0 0 0 0 0
111010: 0 0 0 0 0 0 0 0
111011: 0 0 0 0 0 0 0 0
111100: 0 0 5 705 0 0 0 0
111101: 0 0 0 0 0 0 0 0
111110: 0 0 74 420 0 0 0 0
111111: 0 0 0 0 0 0 0 0

Opcode histogram for ALU1, type 0
000 001 010 011 100 101 110 111
000000: 376 9 7 4 18 0 0 0
000001: 43 39 23 4 0 0 1 31
000010: 7 0 3 2 5 0 0 0
000011: 0 0 11 3 0 0 0 3
000100: 222 5 0 68 0 0 0 0
000101: 370 0 0 25 0 0 0 0
000110: 250 0 0 4 0 0 0 0
000111: 59 0 1 17 0 0 0 0
001000: 3 0 0 3 0 0 0 0
001001: 345 0 0 4 0 0 0 0
001010: 764 0 0 18 0 0 0 0
001011: 372 0 0 9 0 0 0 0
001100: 16 0 0 2 0 0 0 0
001101: 392 0 0 0 0 0 0 0
001110: 6 0 0 2 0 0 0 0
001111: 0 0 0 357 0 0 0 0

010000: 7 0 0 27 0 0 0 0
010001: 41 3 9 205 3 0 0 0
010010: 225 12 26 1782 6 5 6 11
010011: 69 68 33 1533 3 0 0 12
010100: 13 2 0 13 3 0 0 0
010101: 0 19 16 14 3 0 0 0
010110: 0 0 0 0 3 0 0 0
010111: 0 0 0 0 3 0 0 0
011000: 3 0 0 0 0 1 1 1
011001: 133 0 0 0 0 0 0 0
011010: 0 0 0 0 2 0 0 0
011011: 0 0 0 0 2 0 0 0
011100: 1 0 0 0 0 0 0 0
011101: 0 0 0 0 0 0 0 0
011110: 0 0 0 0 0 0 0 0
011111: 0 0 0 0 0 0 0 0
100000: 1104 91 146 15396 19 0 6 11
100001: 2 0 0 0 9 0 0 0
100010: 37 0 0 3 1 0 0 0
100011: 86 5 6 91 9 0 0 0
100100: 297 67 46 1452 55 7 0 289
100101: 1 0 0 0 9 0 0 0
100110: 588 0 0 35 1367 214 35 523
100111: 6425 400 78 3142 19 0 0 16
101000: 356 0 0 8 0 0 0 0
101001: 786 0 0 23706 0 0 0 0
101010: 32 0 0 0 0 0 0 0
101011: 0 0 0 0 0 0 0 0
101100: 64 3 3 714 0 0 0 0
101101: 0 0 0 0 0 0 0 0
101110: 2346 190 70 883 3 0 0 0
101111: 39 0 0 20 0 0 0 0
110000: 0 0 0 0 0 0 0 0
110001: 0 0 0 0 0 0 0 0
110010: 58 0 0 10 0 0 0 0
110011: 136 1 0 27 0 0 0 0
110100: 715 11 2 211 0 0 0 0
110101: 1019 13 2 231 0 0 0 0
110110: 198 0 0 20 0 0 0 0
110111: 131 3 0 39 0 0 0 0
111000: 0 0 0 0 0 0 0 0
111001: 28 0 0 2 0 0 0 0
111010: 0 0 0 0 0 0 0 0
111011: 0 0 0 0 0 0 0 0
111100: 28 0 0 12 0 0 0 0
111101: 113 0 0 30 0 0 0 0
111110: 20 0 0 3 0 0 0 0
111111: 59 0 0 8 0 0 0 0

Opcode histogram for ALU1, type 1
000 001 010 011 100 101 110 111
000000: 5 0 0 0 0 0 0 0
000001: 0 0 0 0 0 0 0 0
000010: 0 0 0 0 0 0 0 0

000011: 0 0 0 0 0 0 0 1
000100: 0 0 0 0 0 0 0 0
000101: 0 0 0 0 0 0 0 0
000110: 0 0 0 0 0 0 0 0
000111: 0 0 0 0 0 0 0 0
001000: 0 0 0 0 0 0 0 0
001001: 0 0 0 0 0 0 0 0
001010: 0 0 0 0 0 0 0 0
001011: 0 0 0 0 0 0 0 0
001100: 0 0 0 0 0 0 0 0
001101: 0 0 0 0 0 0 0 0
001110: 0 0 0 0 0 0 0 0
001111: 0 0 0 0 0 0 0 0
010000: 1 0 0 0 0 0 0 0
010001: 0 0 0 0 0 0 0 0
010010: 0 0 0 0 0 0 0 0
010011: 0 0 0 0 0 0 0 0
010100: 0 0 0 0 0 0 0 0
010101: 0 0 0 0 0 0 0 0
010110: 0 0 0 0 0 0 0 0
010111: 0 0 0 0 0 0 0 0
011000: 0 0 0 0 0 0 0 0
011001: 0 0 0 0 0 0 0 0
011010: 0 0 0 0 0 0 0 0
011011: 0 0 0 0 0 0 0 0
011100: 0 0 0 0 0 0 0 0
011101: 0 0 0 0 0 0 0 0
011110: 0 0 0 0 0 0 0 0
011111: 0 0 0 0 0 0 0 0
100000: 0 0 0 0 0 0 0 0
100001: 0 0 0 17 0 0 0 0
100010: 0 0 0 2 0 0 0 0
100011: 0 0 0 0 0 0 0 0
100100: 487 0 0 0 8 0 0 0
100101: 8 0 0 0 4 0 0 0
100110: 0 0 0 0 0 0 0 0
100111: 0 0 0 0 0 0 0 0
101000: 227 0 0 0 25 0 0 0
101001: 208 0 0 0 2 0 0 0
101010: 122 0 0 0 14 0 0 0
101011: 85 0 0 0 2 0 0 0
101100: 15 0 0 0 1 0 0 0
101101: 1 0 0 0 1 0 0 0
101110: 0 0 0 0 0 0 0 0
101111: 0 0 0 0 0 0 0 0
110000: 0 0 0 0 0 0 0 0
110001: 2 0 0 0 0 0 0 0
110010: 0 0 0 0 0 0 0 0
110011: 0 0 0 0 1 0 0 0
110100: 0 0 0 0 1 0 0 0
110101: 11 0 0 0 1 0 0 0
110110: 56 0 0 0 3 0 0 0
110111: 1 0 0 0 1 0 0 0
111000: 0 0 0 0 1 0 0 0

111001: 0 0 0 0 1 0 0 0
111010: 3 0 0 0 3 0 0 0
111011: 8 0 0 0 1 0 0 0
111100: 0 0 0 0 0 0 0 0
111101: 3 0 0 0 1 0 0 0
111110: 0 0 0 0 0 0 0 0
111111: 0 0 0 0 0 0 0 0

Opcode histogram for ALU1, type 2
000 001 010 011 100 101 110 111
000000: 1 0 0 0 0 0 0 0
000001: 0 0 0 0 0 0 0 0
000010: 0 0 0 0 0 0 0 0
000011: 0 0 0 0 0 0 0 0
000100: 0 0 0 0 0 0 0 0
000101: 0 0 0 0 0 0 0 0
000110: 0 0 0 0 0 0 0 0
000111: 0 0 0 0 0 0 0 0
001000: 16 0 0 0 0 0 0 0
001001: 33 0 0 0 0 0 0 0
001010: 270 0 0 0 0 0 0 0
001011: 137 0 0 0 0 0 0 0
001100: 1523 0 0 0 0 0 0 0
001101: 1587 0 0 30 0 0 0 0
001110: 0 0 0 0 0 0 0 0
001111: 2 0 0 0 0 0 0 0
010000: 0 0 0 0 0 0 0 0
010001: 0 0 0 0 0 0 0 0
010010: 0 0 0 0 0 0 0 0
010011: 0 0 0 0 0 0 0 0
010100: 0 0 0 0 0 0 0 0
010101: 0 0 0 0 21 0 0 19
010110: 5 0 0 0 0 0 0 0
010111: 3 0 0 0 0 0 0 0
011000: 0 0 0 0 0 0 0 0
011001: 0 0 0 0 0 0 0 0
011010: 0 0 0 0 0 0 0 0
011011: 0 0 0 0 0 0 0 0
011100: 477 0 0 0 0 0 0 0
011101: 192 0 0 39 0 0 0 0
011110: 3 0 0 0 0 0 0 0
011111: 0 0 0 0 0 0 0 0
100000: 18 0 0 0 0 0 0 0
100001: 1 0 0 0 0 0 0 0
100010: 16 0 0 0 0 0 0 0
100011: 2 0 0 0 0 0 0 0
100100: 0 0 0 0 0 0 0 0
100101: 1 0 0 0 0 0 0 0
100110: 0 0 0 0 0 0 0 0
100111: 12 0 0 0 0 0 0 0
101000: 0 0 0 0 0 0 0 0
101001: 0 0 0 0 0 0 0 0
101010: 0 0 0 0 0 0 0 0
101011: 0 0 0 0 0 0 0 0

101100: 73 0 0 0 0 0 0 0
101101: 100 0 0 4 0 0 0 0
101110: 0 0 0 0 0 0 0 0
101111: 11 0 0 112 0 0 0 0
110000: 0 0 0 0 0 0 0 0
110001: 0 0 0 0 0 0 0 0
110010: 0 0 0 0 0 0 0 0
110011: 0 0 0 0 0 0 0 0
110100: 0 0 0 0 0 0 0 0
110101: 0 0 0 0 0 0 0 0
110110: 0 0 0 0 0 0 0 0
110111: 0 0 0 0 0 0 0 0
111000: 0 0 0 0 0 0 0 0
111001: 0 0 0 0 0 0 0 0
111010: 0 0 0 0 0 0 0 0
111011: 0 0 0 0 0 0 0 0
111100: 253 0 0 0 0 0 0 0
111101: 507 0 0 20 0 0 0 0
111110: 0 0 0 0 0 0 0 0
111111: 6 0 0 0 0 0 0 0

Opcode histogram for ALU1, type 3
000 001 010 011 100 101 110 111
000000: 0 0 0 0 0 0 0 0
000001: 0 0 0 0 0 0 0 0
000010: 0 0 0 0 0 0 0 0
000011: 0 0 0 0 0 0 0 0
000100: 0 0 0 0 0 0 0 0
000101: 0 0 0 0 0 0 0 0
000110: 0 0 0 0 0 0 0 0
000111: 0 0 0 0 0 0 0 0
001000: 0 0 0 0 1 0 0 0
001001: 0 0 0 0 1 0 0 0
001010: 0 0 0 0 1 0 0 0
001011: 0 0 0 0 1 0 0 0
001100: 0 0 0 0 1 0 0 0
001101: 0 0 0 0 1 0 0 0
001110: 0 0 0 0 1 0 0 0
001111: 0 0 0 0 1 0 0 0
010000: 0 0 0 0 11 0 0 0
010001: 0 0 0 0 0 0 0 0
010010: 0 0 0 0 1 0 0 0
010011: 0 0 0 0 1 0 0 0
010100: 0 0 0 0 0 0 0 0
010101: 0 0 0 0 1 0 0 0
010110: 0 0 0 0 0 0 0 0
010111: 0 0 0 0 0 0 0 0
011000: 0 0 0 0 0 0 0 0
011001: 0 0 0 0 0 0 0 0
011010: 0 0 0 0 0 0 0 0
011011: 0 0 0 0 0 0 0 0
011100: 0 0 0 0 0 0 0 0
011101: 0 0 0 0 0 0 0 0
011110: 0 0 0 0 0 0 0 0

011111: 0 0 0 0 0 0 0 0
100000: 7 0 0 0 0 0 0 0
100001: 0 0 0 78 0 0 0 0
100010: 0 0 0 0 0 0 0 0
100011: 0 0 0 0 0 0 0 0
100100: 6 0 0 0 0 0 0 0
100101: 387 0 0 0 0 0 0 0
100110: 0 0 0 4 0 0 0 0
100111: 1 0 0 0 0 0 0 0
101000: 43 0 0 2 0 0 0 0
101001: 0 0 0 11 0 0 0 0
101010: 0 0 0 0 0 0 0 0
101011: 0 0 0 0 0 0 0 0
101100: 0 0 0 0 0 0 0 0
101101: 0 0 0 0 0 0 0 0
101110: 0 0 0 0 0 0 0 0
101111: 0 0 0 0 0 0 0 0
110000: 0 0 0 0 0 0 0 0
110001: 0 0 0 0 0 0 0 0
110010: 0 0 0 0 0 1 0 0
110011: 1 0 0 0 0 0 0 0
110100: 0 0 0 0 0 0 0 0
110101: 0 0 0 0 0 0 0 0
110110: 0 0 0 0 0 0 0 0
110111: 0 0 0 0 0 0 0 0
111000: 14 0 0 0 0 0 0 0
111001: 0 0 0 0 0 0 0 0
111010: 0 0 0 0 0 0 0 0
111011: 0 0 0 0 0 0 0 0
111100: 0 0 0 0 0 0 0 0
111101: 0 0 0 0 0 0 0 0
111110: 0 0 0 0 0 0 0 0
111111: 0 0 0 0 0 0 0 0

Opcode histogram for ALU1, type 4
000 001 010 011 100 101 110 111
000000: 21 0 0 0 0 0 0 0
000001: 3 0 0 0 0 0 0 0
000010: 2 0 0 0 0 0 0 0
000011: 2 0 0 0 0 0 0 0
000100: 2 0 0 0 0 0 0 0
000101: 1 0 0 0 0 0 0 0
000110: 1 0 0 0 0 0 0 0
000111: 0 0 0 0 0 0 0 0
001000: 0 0 0 0 0 0 0 0
001001: 3 0 7 0 0 0 0 0
001010: 2 0 0 0 0 0 0 0
001011: 2 0 0 0 0 0 0 0
001100: 136 0 0 0 0 0 0 0
001101: 3 0 0 0 0 0 0 0
001110: 0 0 0 0 0 0 0 0
001111: 0 0 0 0 0 0 0 0
010000: 3 0 0 0 0 0 0 0
010001: 3 0 0 0 0 0 0 0

010010: 0 0 0 0 0 0 0 0
010011: 0 0 0 0 0 0 0 0
010100: 0 0 0 0 0 0 0 0
010101: 0 0 0 0 0 0 0 0
010110: 0 0 0 0 0 0 0 0
010111: 0 0 0 0 0 0 0 0
011000: 0 0 0 0 0 0 0 0
011001: 0 0 0 0 0 0 0 0
011010: 0 0 0 0 0 0 0 0
011011: 0 0 0 0 0 0 0 0
011100: 0 0 0 0 0 0 0 0
011101: 0 0 0 0 0 0 0 0
011110: 0 0 0 0 0 0 0 0
011111: 0 0 0 0 0 0 0 0
100000: 3 0 0 0 0 0 0 0
100001: 3 0 0 0 0 0 0 0
100010: 2 0 0 0 0 0 0 0
100011: 1 0 0 0 0 0 0 0
100100: 0 0 0 0 0 0 0 0
100101: 0 0 0 0 0 0 0 0
100110: 0 0 0 0 0 0 0 0
100111: 0 0 0 0 0 0 0 0
101000: 4 0 0 0 0 0 0 0
101001: 4 0 0 0 0 0 0 0
101010: 4 0 0 0 0 0 0 0
101011: 0 0 0 0 0 0 0 0
101100: 0 0 0 0 0 0 0 0
101101: 0 0 0 0 0 0 0 0
101110: 0 0 0 0 0 0 0 0
101111: 0 0 0 0 0 0 0 0
110000: 0 0 0 0 0 0 0 0
110001: 0 0 0 0 0 0 0 0
110010: 0 0 0 0 0 0 0 0
110011: 0 0 0 0 0 0 0 0
110100: 0 0 0 0 0 0 0 0
110101: 0 0 0 0 0 0 0 0
110110: 0 0 0 0 0 0 0 0
110111: 0 0 0 0 0 0 0 0
111000: 0 0 0 0 0 0 0 0
111001: 0 0 0 0 0 0 0 0
111010: 0 0 0 0 0 0 0 0
111011: 0 0 0 0 0 0 0 0
111100: 0 0 0 0 0 0 0 0
111101: 0 0 0 0 0 0 0 0
111110: 0 0 0 0 0 0 0 0
111111: 0 0 0 0 0 0 0 0

Opcode histogram for ALU1, type 5
000 001 010 011 100 101 110 111
000000: 0 0 0 0 0 0 0 0
000001: 0 0 0 0 0 0 0 0
000010: 0 0 0 0 0 0 0 0
000011: 0 0 0 0 0 0 0 0
000100: 0 0 0 0 0 0 0 0

000101: 0 0 0 0 0 0 0 0
000110: 0 0 0 0 0 0 0 0
000111: 0 0 0 0 0 0 0 0
001000: 0 0 0 0 0 0 0 0
001001: 0 0 0 0 0 0 0 0
001010: 0 0 0 0 0 0 0 0
001011: 0 0 0 0 0 0 0 0
001100: 0 0 0 0 0 0 0 0
001101: 0 0 0 0 0 0 0 0
001110: 0 0 0 0 0 0 0 0
001111: 0 0 0 0 0 0 0 0
010000: 7 0 0 0 0 0 0 0
010001: 0 0 0 0 0 0 0 0
010010: 0 0 0 0 0 0 0 0
010011: 0 0 0 0 0 0 0 0
010100: 3 0 0 0 0 0 0 0
010101: 0 0 0 0 0 0 0 0
010110: 2 0 0 0 0 0 0 0
010111: 0 0 0 0 0 0 0 0
011000: 1 0 0 0 0 0 0 0
011001: 0 0 0 0 0 0 0 0
011010: 10 0 0 0 0 0 0 0
011011: 0 0 0 0 0 0 0 0
011100: 0 0 0 0 0 0 0 0
011101: 0 0 0 0 0 0 0 0
011110: 1 0 0 0 0 0 0 0
011111: 0 0 0 0 0 0 0 0
100000: 0 0 0 0 0 0 0 0
100001: 0 0 0 0 0 0 0 0
100010: 0 0 0 0 0 0 0 0
100011: 0 0 0 0 0 0 0 0
100100: 0 0 0 0 0 0 0 0
100101: 0 0 0 0 0 0 0 0
100110: 0 0 0 0 0 0 0 0
100111: 0 0 0 0 0 0 0 0
101000: 2 0 0 0 0 0 0 0
101001: 2 0 0 0 0 0 0 0
101010: 0 0 0 0 0 0 0 0
101011: 0 0 0 0 0 0 0 0
101100: 0 0 0 0 0 0 0 0
101101: 0 0 0 0 0 0 0 0
101110: 0 0 0 0 0 0 0 0
101111: 0 0 0 0 0 0 0 0
110000: 0 0 0 0 0 0 0 0
110001: 0 0 0 0 0 0 0 0
110010: 0 0 0 0 0 0 0 0
110011: 0 0 0 0 0 0 0 0
110100: 0 0 0 0 0 0 0 0
110101: 0 0 0 0 0 0 0 0
110110: 0 0 0 0 0 0 0 0
110111: 0 0 0 0 0 0 0 0
111000: 0 0 0 0 0 0 0 0
111001: 0 0 0 0 0 0 0 0
111010: 0 0 0 0 0 0 0 0

111011: 0 0 0 0 0 0 0 0
111100: 1 0 0 0 0 0 0 0
111101: 0 0 0 0 0 0 0 0
111110: 0 0 0 0 0 0 0 0
111111: 0 0 0 0 0 0 0 0

Opcode histogram for ALU1, type 6
000 001 010 011 100 101 110 111
000000: 1 0 0 0 0 0 0 0
000001: 0 0 0 0 0 0 0 0
000010: 0 0 0 0 0 0 0 0
000011: 0 0 0 0 0 0 0 0
000100: 0 0 0 0 0 0 0 0
000101: 0 0 0 0 0 0 0 0
000110: 0 0 0 0 0 0 0 0
000111: 0 0 0 0 0 0 0 0
001000: 0 0 0 0 0 0 0 0
001001: 0 0 0 0 0 0 0 0
001010: 0 0 0 0 0 0 0 0
001011: 0 0 0 0 0 0 0 0
001100: 0 0 0 0 0 0 0 0
001101: 0 0 0 0 0 0 0 0
001110: 0 0 0 0 0 0 0 0
001111: 0 0 0 0 0 0 0 0
010000: 0 0 0 0 0 0 0 0
010001: 0 0 0 0 0 0 0 0
010010: 0 0 0 0 0 0 0 0
010011: 0 0 0 0 0 0 0 0
010100: 0 0 0 0 0 0 0 0
010101: 0 0 0 0 0 0 0 0
010110: 0 0 0 0 0 0 0 0
010111: 0 0 0 0 0 0 0 0
011000: 0 0 0 0 0 0 0 0
011001: 0 0 0 0 0 0 0 0
011010: 0 0 0 0 0 0 0 0
011011: 0 0 0 0 0 0 0 0
011100: 0 0 0 0 0 0 0 0
011101: 0 0 0 0 0 0 0 0
011110: 0 0 0 0 0 0 0 0
011111: 0 0 0 0 0 0 0 0
100000: 2 0 0 0 0 0 0 0
100001: 0 0 0 0 0 0 0 0
100010: 0 0 0 0 0 0 0 0
100011: 0 0 0 0 0 0 0 0
100100: 0 0 0 0 0 0 0 0
100101: 0 0 0 0 0 0 0 0
100110: 0 0 0 0 0 0 0 0
100111: 0 0 0 0 0 0 0 0
101000: 0 0 0 0 0 0 0 0
101001: 0 0 0 0 0 0 0 0
101010: 0 0 0 0 0 0 0 0
101011: 0 0 0 0 0 0 0 0
101100: 0 0 0 0 0 0 0 0
101101: 0 0 0 0 0 0 0 0

101110: 0 0 0 0 0 0 0 0
101111: 0 0 0 0 0 0 0 0
110000: 0 0 0 0 0 0 0 0
110001: 0 0 0 0 0 0 0 0
110010: 0 0 0 0 0 0 0 0
110011: 0 0 0 0 0 0 0 0
110100: 0 0 0 0 0 0 0 0
110101: 0 0 0 0 0 0 0 0
110110: 0 0 0 0 0 0 0 0
110111: 0 0 0 0 0 0 0 0
111000: 0 0 0 0 0 0 0 0
111001: 0 0 0 0 0 0 0 0
111010: 0 0 0 0 0 0 0 0
111011: 0 0 0 0 0 0 0 0
111100: 0 0 0 0 0 0 0 0
111101: 0 0 0 0 0 0 0 0
111110: 0 0 0 0 0 0 0 0
111111: 0 0 0 0 0 0 0 0

Opcode histogram for ALU1, type 7
000 001 010 011 100 101 110 111
000000: 2 0 0 0 0 0 0 0
000001: 0 0 0 0 0 0 0 0
000010: 0 0 0 0 0 0 0 0
000011: 0 0 0 0 0 0 0 0
000100: 0 0 0 0 0 0 0 0
000101: 0 0 0 0 0 0 0 0
000110: 0 0 0 0 0 0 0 0
000111: 0 0 0 0 0 0 0 0
001000: 0 0 0 0 0 0 0 0
001001: 0 0 0 0 0 0 0 0
001010: 0 0 0 0 0 0 0 0
001011: 0 0 0 0 0 0 0 0
001100: 0 0 0 0 0 0 0 0
001101: 0 0 0 0 0 0 0 0
001110: 0 0 0 0 0 0 0 0
001111: 0 0 0 0 0 0 0 0
010000: 0 0 0 0 0 0 0 0
010001: 0 0 0 0 0 0 0 0
010010: 0 0 0 0 0 0 0 0
010011: 0 0 0 0 0 0 0 0
010100: 0 0 0 0 0 0 0 0
010101: 0 0 0 0 0 0 0 0
010110: 0 0 0 0 0 0 0 0
010111: 0 0 0 0 0 0 0 0
011000: 0 0 0 0 0 0 0 0
011001: 0 0 0 0 0 0 0 0
011010: 0 0 0 0 0 0 0 0
011011: 0 0 0 0 0 0 0 0
011100: 0 0 0 0 0 0 0 0
011101: 0 0 0 0 0 0 0 0
011110: 0 0 0 0 0 0 0 0
011111: 0 0 0 0 0 0 0 0
100000: 2 0 0 0 0 0 0 0

100001: 0 0 0 0 0 0 0 0
100010: 0 0 0 0 0 0 0 0
100011: 0 0 0 0 0 0 0 0
100100: 0 0 0 0 0 0 0 0
100101: 0 0 0 0 0 0 0 0
100110: 0 0 0 0 0 0 0 0
100111: 0 0 0 0 0 0 0 0
101000: 0 0 0 0 0 0 0 0
101001: 0 0 0 0 0 0 0 0
101010: 0 0 0 0 0 0 0 0
101011: 0 0 0 0 0 0 0 0
101100: 0 0 0 0 0 0 0 0
101101: 0 0 0 0 0 0 0 0
101110: 0 0 0 0 0 0 0 0
101111: 0 0 0 0 0 0 0 0
110000: 0 0 0 0 0 0 0 0
110001: 0 0 0 0 0 0 0 0
110010: 0 0 0 0 0 0 0 0
110011: 0 0 0 0 0 0 0 0
110100: 0 0 0 0 0 0 0 0
110101: 0 0 0 1 0 0 0 0
110110: 0 0 0 0 0 0 0 0
110111: 0 0 0 0 0 0 0 0
111000: 0 0 0 0 0 0 0 0
111001: 0 0 0 0 0 0 0 0
111010: 0 0 0 0 0 0 0 0
111011: 0 0 0 0 0 0 0 0
111100: 0 0 0 0 0 0 0 0
111101: 0 0 0 0 0 0 1 0
111110: 0 0 0 0 0 0 0 0
111111: 0 0 0 0 0 0 0 6

Opcode histogram for LSU
000 001 010 011 100 101 110 111
000000: 61 0 0 0 0 0 0 0
000001: 75 0 0 0 0 0 0 0
000010: 81 0 0 0 0 0 0 1
000011: 8 0 0 0 6 0 0 0
000100: 0 1 0 0 83 0 0 0
000101: 0 0 0 0 250 0 0 0
000110: 0 0 0 0 0 0 0 0
000111: 0 0 2 11903 0 0 0 0
001000: 827 0 2 549 9946 14 98 2411
001001: 0 0 0 0 0 0 0 0
001010: 14916 16 117 3330 228 0 0 0
001011: 315 0 0 0 241 0 0 0
001100: 8 0 0 0 1 0 0 0
001101: 0 0 0 0 0 0 0 0
001110: 0 0 0 0 0 0 0 0
001111: 15 0 0 3 0 0 0 0
010000: 310 0 2 0 2 0 0 0
010001: 26 12 227 54 6 0 12 134
010010: 905 15 0 18 817 0 0 0
010011: 48 0 1 0 68 0 1 1

010100: 4 0 0 0 0 0 0 0
010101: 0 0 0 0 0 0 0 0
010110: 2 57 14 0 36 47 13 0
010111: 50 30 0 0 3 36 0 0
011000: 26 0 0 0 82 0 0 0
011001: 0 0 0 0 0 0 0 0
011010: 18 0 0 0 18 0 0 0
011011: 18 0 0 0 18 0 0 0
011100: 0 0 0 0 15 0 0 0
011101: 0 0 0 0 0 0 0 0
011110: 0 0 0 0 0 0 0 0
011111: 0 0 0 0 0 0 0 0
100000: 0 0 0 2 0 0 0 0
100001: 0 0 0 0 0 0 0 0
100010: 1 0 0 1 0 0 0 0
100011: 1 0 0 0 0 0 0 0
100100: 0 0 0 0 0 0 0 0
100101: 0 0 0 0 0 0 0 0
100110: 0 0 0 1 0 0 0 0
100111: 5 0 0 0 0 0 0 0
101000: 1 0 0 1 0 0 0 0
101001: 6 0 0 9 0 0 0 0
101010: 0 0 0 0 0 0 0 0
101011: 0 0 0 0 0 0 0 0
101100: 0 0 0 1 0 0 0 0
101101: 0 0 0 0 0 0 0 0
101110: 0 0 0 1 0 0 0 0
101111: 0 0 0 0 0 0 0 0
110000: 0 0 0 1 0 0 0 0
110001: 0 0 0 0 0 0 0 0
110010: 0 0 0 0 0 0 0 0
110011: 0 0 0 0 0 0 0 0
110100: 0 0 0 2 0 0 0 0
110101: 0 0 0 0 0 0 0 0
110110: 0 0 0 0 0 0 0 0
110111: 0 0 0 8 0 0 0 0
111000: 0 0 0 0 0 0 0 0
111001: 0 0 0 0 0 0 0 0
111010: 0 0 0 0 0 0 0 0
111011: 0 0 0 0 0 0 0 0
111100: 0 0 0 2 0 0 0 0
111101: 0 0 0 0 0 0 0 0
111110: 0 0 0 1 0 0 0 0
111111: 0 0 0 0 0 0 0 0